GD2S02: Software Engineering for Games

Design Patterns – Games

Outline

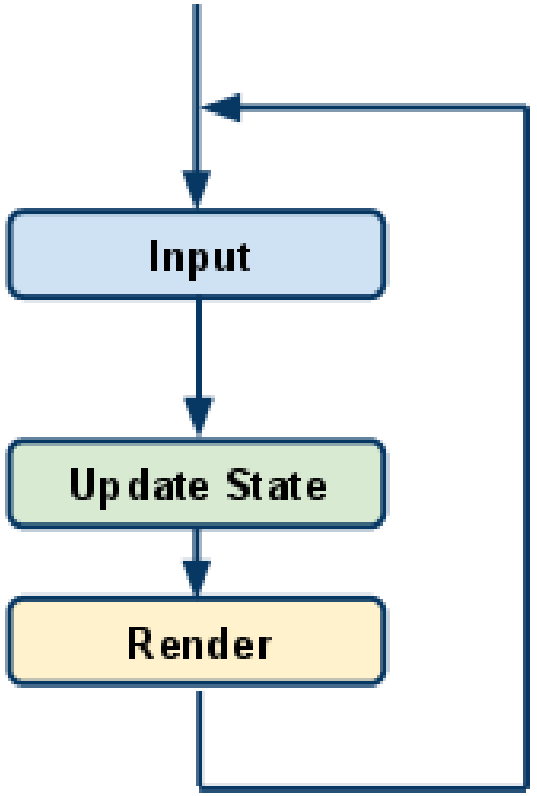
- Game Design patterns
  - Sequence patterns
    - Game loop
    - Update Method
    - Double buffer
  - Decoupling Patterns
    - Component
  - Behavioral patterns
    - Type object
    - Subclass sandbox
    - Bytecode/Virtual machine

Sequence Patterns

- One aspect that most game worlds feature is *time*—the artificial world lives and breathes at its own cadence.
- As world builders, we must invent time and craft the gears that drive our game's great clock.
- Sequencing patterns are tools for doing that.
  - A Game Loop is the central axle that the clock spins on.
  - Objects hear its ticking through Update Methods.
  - We can hide the computer's sequential nature behind a facade of snapshots of moments in time using Double Buffering so that the world appears to update simultaneously.

Game Loop

- Interactive real time systems have three main modules:
  - 1- Input data acquisition
  - 2- Processing
  - 3- Presentation of results.
- A game loop runs continuously during gameplay. Each turn of the loop, it processes user input **without blocking**, updates the game state, and renders the game. It tracks the passage of time to control the rate of gameplay.



Source: https://www.ssugames.org/pluginfile.php/1026/mod_resource/content/1/2005_sbgames.pdf,
Image: http://obviam.net/index.php/a-very-basic-the-game-loop-for-android/

MEDIA DESIGN SCHOOL GAME DEV

Game Loop

- The first key part of a real game loop: *it processes user input, but doesn't wait for it*

```
while (true)
{
        processInput();
        updateGameState();
        displayGameState();
}
```

- The second key part is : *it runs the game at a consistent speed despite differences in the underlying hardware.*

Game Loop

- The Game Loop may vary by game type.
- However, the same can be achieved by the game maintaining state and using the simple game loop.
- The simple game loop does not describe
  - Separation of stages
  - Uncoupled stages
  - Multi-threading

Source: http://obviam.net/index.php/the-android-game-loop/

Update Method

- Simulates a collection of independent objects by telling each to process one frame of behavior at a time.
- *Each entity in the game should encapsulate its own behavior.* This will keep the game loop uncluttered and make it easy to add and remove entities.
- The **game world** maintains a **collection of objects**. Each object implements an **update method** that **simulates one frame** of the object's behavior. Each frame, the game updates every object in the collection.

Update Method

- Update methods work well when:
  - Your game has a number of objects or systems that need to run simultaneously.
  - Each object's behavior is mostly independent of the others.
  - The objects need to be simulated over time.

Code Example

```
class Entity {
public:
  Entity()  : x_(0), y_(0)   {}

  virtual ~Entity() {}
  virtual void update() = 0;
  double x() const { return x_; }
  double y() const { return y_; }

  void setX(double x) { x_ = x; }
  void setY(double y) { y_ = y; }

private:
  double x_;
  double y_;
};

class World {
public:
  World()  : numEntities_(0)   {}
  void gameLoop();

private:
  Entity* entities_[MAX_ENTITIES];
  int numEntities_;
};
```

```
void World::gameLoop()
{
  while (true)
  {
    // Handle user input...
   processInput();
    // Update each entity.
    for (int i = 0; i < numEntities_; i++)
    {
      entities_[i]->update();
    }
    // Physics and rendering...
   displayGameState();
  }
}
```

Double Buffer

- Cause a series of sequential operations to appear instantaneous or simultaneous.
- A framebuffer is used in most computers to store information about the colors of each pixel on screen.
- Double buffers means we have two framebuffers. The first represents the current frame the video hardware is reading from. The other is where the rendering code is writing. So when the rendering code finishes drawing the scene, it switches to the second buffer and so no.

Double Buffer

- Not just for graphics.
- The core problem that a Double Buffer solves is state being accessed while it's being modified.
- Another equally common cause: when the code *doing the modification* is accessing the same state that it is modifying.
- This can manifest in places like physics and AI, where you have entities interacting with each other.

Double Buffer

- A **buffered class** encapsulates a **buffer**: a piece of state that can be modified. The class keeps *two* instances of the buffer: a **next buffer** and a **current buffer**.
- Information is read *from* the *current* buffer.
- Information is written *to* the *next* buffer.
- When the changes are complete, a **swap** operation swaps the next and current buffers instantly.

Double Buffer

- This pattern is appropriate when these things are true:
  - We have some state that is being modified incrementally.
  - That same state may be accessed in the middle of modification.
  - We want to prevent the code that's accessing the state from seeing the work-in-progress.
  - We want to be able to read the state and we don't want to have to wait while it's being written.

Double Buffer Consequences

- Increased Memory usage, because we need to keep two copies of the buffer in memory all the time.
- The swap step needs to be atomic, no code can access either buffers while they are being swapped.

Component

- Allow a single class to bridge multiple domains without coupling the domains to each other.
- Slice your design into separate parts (components) along domain boundaries, with a thin shell to bind all the components together.

Component

- This pattern can be used when any of these is true:
  - You have a class that touches multiple domains which you want to keep decoupled from each other.
  - A class is getting massive and hard to work with.
  - You want to be able to define a variety of objects that share different capabilities, but using inheritance does not let you pick just the parts you want to reuse precisely enough.

Component

- Example: Player class in a platformer
  - Reading controller input and translating that to motion.
  - Interaction with the level, so some physics and collision.
  - He has got to show up on screen, so toss in animation and rendering.
  - He will probably play some sounds too.

Component

- We will take our monolithic class and slice it into separate parts along domain boundaries
- For example, we will take the code for handling user input and move it into a separate *InputComponent.*
- Repeat for other components.
- All that remains is a thin shell that binds the components together.

Component

- Our component classes are now decoupled.
- Even though the player class has a *PhysicsComponent* and a *GraphicsComponent*, the two do not know about each other.
- This means the person working on physics can modify their component without needing to know anything about graphics and vice versa

Type Object

- Allow the flexible creation of new “classes” by creating a single class, each instance of which represents a different type of object.
- Define a type object class and a typed object class. Each type object instance represents a different logical type. Each typed object stores a reference to the type object that describes its type.
- The Type Object pattern lets us build a type system as if we were designing our own programming language.

Type Object

- Imagine we are working on a fantasy role-playing game.
- Our task is to write the code for the hordes of vicious monsters that seek to slay our brave hero.
- Monsters have a bunch of different attributes: health, attacks, etc
- The classic OOP design follows…

Type Object

```
class Monster
{
public:
  virtual const char* getAttack() =
    0;

protected:
  Monster(int startingHealth)
  : health_(startingHealth)
  {}

private:
  int health_; // Current health.
};
```

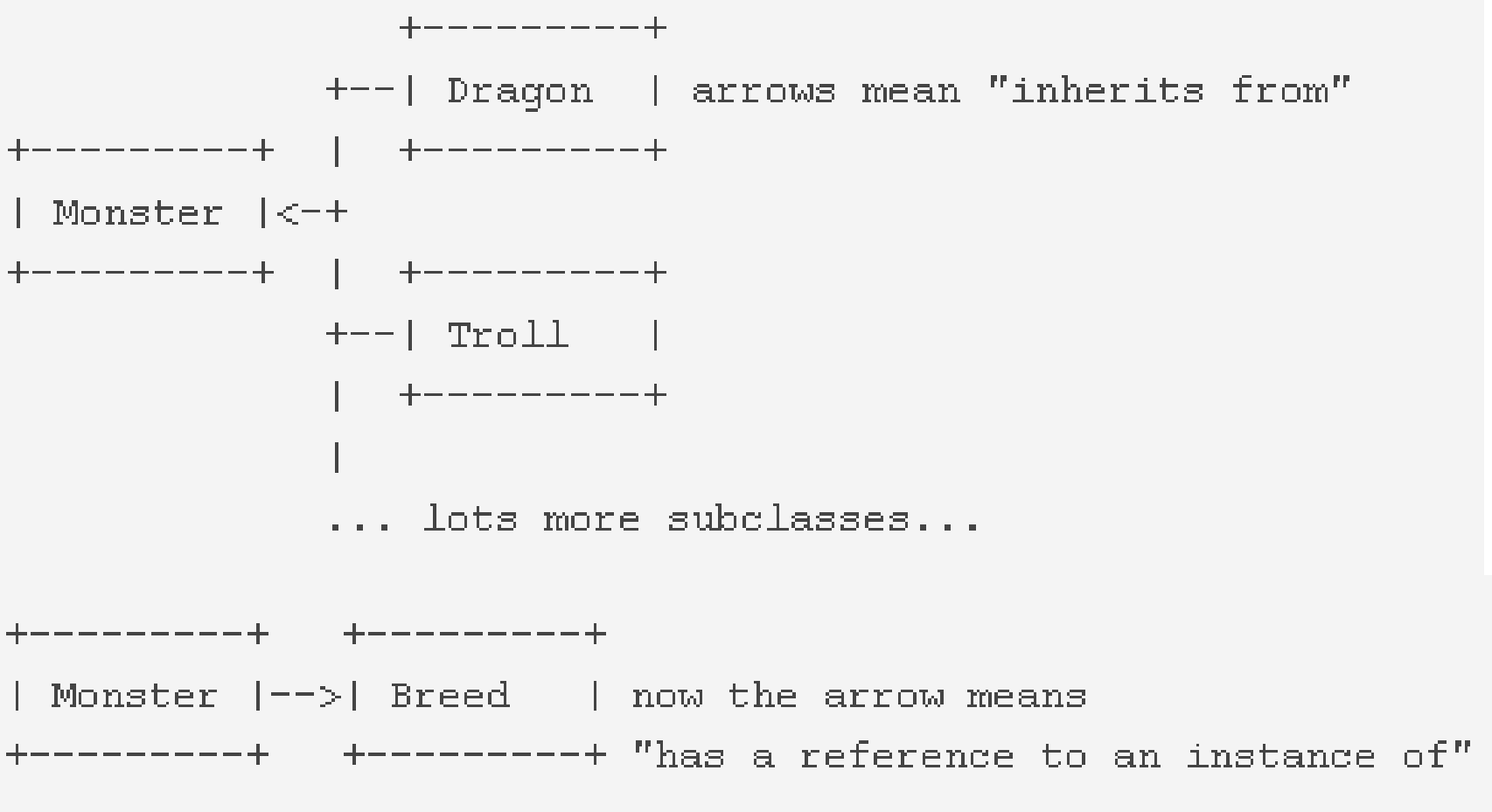
```
class Dragon : public Monster
{
public:
  Dragon() : Monster(230) {}
  const char* getAttack() { return "The
     dragon breathes fire!"; }
};

class Troll : public Monster
{
public:
  Troll() : Monster(48) {}
  const char* getAttack() { return "The
     troll clubs you!"; }
};
```

Type Object

- Then, strangely, things start to bog down
- Our designers ultimately want to have *hundreds* of breeds, and we find ourselves spending all of our time writing these little seven line subclasses and recompiling.

Type Object



MEDIA DESIGN SCHOOL GAME DEV

Type Object

```
class Breed
{
public:
  Breed(int health, const char* attack)
  : health_(health),
    attack_(attack)
  {}

  int   getHealth() { return health_; }
  const char* getAttack() { return attack_; }

private:
  int         health_; // Starting health.
  const char* attack_;
};
```

Type Object

```
class Monster
{
public:
  Monster(Breed& breed)
  : health_(breed.getHealth()),
     breed_(breed)
  {}

  const char* getAttack()
   { return breed_.getAttack(); }

private:
  int    health_; // Current health.
  Breed& breed_;
};
```

Typed Object

Type Object

- When to use it:
  - You don't know what types you will need up front.

    (For example, what if our game needed to support downloading content that contained new breeds of monsters?)
  - You want to be able to modify or add new types without having to recompile or change code.

Subclass Sandbox

- Define behavior in a subclass using a set of operations provided by its base class.
- Create varied behavior while minimising coupling by defining subclasses that only use operations provided by their shared base class.

Subclass Sandbox

- The **base class** defines an abstract **sandbox method**, and several **provided operations**
- Marking them protected makes it clear to a user that they are for use by derived classes
- Each derived **sandboxed subclass** implements the sandbox method using the provided operations

Subclass Sandbox

```
class Superpower
{
protected:
  virtual void activate() = 0;

  void move(float x, float y, float z)
  {
    // Code here...
  }
  void playSound(SoundId sound, float volume)
  {
    // Code here...
  }
  void spawnParticles(ParticleType type, int
      count)
  {
    // Code here...
  }
};
```

```
class SkyLaunch : public Superpower
{
protected:
  virtual void activate()
  {
    // Spring into the air.
    playSound(SOUND_SPROING, 1.0f);
    spawnParticles(PARTICLE_DUST,
     10);
    move(0, 0, 20);
  }
};
```

Subclass Sandbox

- *A Subclass Sandbox is a good fit when:*
  - *You have a base class with a number of derived classes.*
  - *The base class is able to provide all of the operations that a derived class may need to perform.*
  - *There is behavioral overlap in the subclasses and you want to make it easier to share code between them.*
  - *You want to minimize coupling between those derived classes and the rest of the program*

Virtual Machine/Bytecode

- Give behavior the flexibility of data by encoding it as instructions for a virtual machine.
- If we can define our behavior in separate data files that the game engine loads and “executes” in some way, we can achieve all of our goals.
- Create a simple and safe imbedded language with which game designers and (potentially) end-users can build game logic.

Virtual Machine/Bytecode

- We define our own *virtual* machine code, we then write a little emulator for it in our game. It would be similar to machine code—dense, linear, relatively low-level—but would also be handled entirely by our game so we could safely sandbox it.
- The little emulator is called a *virtual machine* (or "VM" for short), and the synthetic binary machine code it runs *bytecode*.
- An **instruction set** defines the low-level operations that can be performed. A series of instructions is encoded as a **sequence of bytes**. A **virtual machine** executes these instructions one at a time, using a **stack for intermediate values**. By combining instructions, complex high-level behavior can be defined.

Virtual Machine/Bytecode

- When to use it:
  - Use it when you have a lot of behavior you need to define and your game's implementation language isn't a good fit because:
    - It's too low-level, making it tedious or error-prone to program in.
    - Iterating on it takes too long due to slow compile times or other tooling issues.
    - It has too much trust. If you want to ensure the behavior being defined can't break the game, you need to sandbox it from the rest of the codebase.